A quick guide to speeding up R code

Rafal Baranowski

LSE 2015, 3 February

Introduction

- Why R is slow?
- Why R is not necessarily slow?
- Possible ways of speeding up R code
  - vectorisation,
  - preallocation of the memory,
  - using optimised R packages (e.g. RcppEigen for solving linear systems),
  - using R byte code compiler,
  - **running R computations in parallel**,
  - **code profiling**,
  - **writing R extensions in C or C++**,
  - **using optimised system libraries**,
  - ...

Estimating volatility with Exponentially Weighted Exponential Average (EWEA)

Let X_t, $t = 1, \ldots, n$ be a univariate time series with $EX_t = 0$ and $\sigma_t^2 = EX_t^2 < \infty$. Let $\lambda \in (0, 1)$. We consider

$$\hat{\sigma}_1 = X_1^2,$$
$$\hat{\sigma}_t = \lambda X_t^2 + (1 - \lambda)\sigma_t^2, \quad , t = 2, \ldots, n$$

Our goal is to **quickly compute** $\hat{\sigma}_t$ for MC time series, where both n and MC are large.

EWEA: R only code

The exponential smoother is straightforward to code:

```
#X is a n by MC matrix, lambda is a scalar
vol.exp.smooth.r <- function(X, lambda, ...){
  apply(X, 2, function(x){
    n <- length(x)
    vol <- rep(0, n) #preallocate memory if you can!
    vol[1] <- x[1]^2
    for(j in 2:n) vol[j] <-  lambda * x[j]^2 + (1-lambda) * vol[j-1]
    return(vol)
  })
}
```

EWEA: Execution time of the R only code

Running

```
n <- 2000 #length of each vector
lambda <- 0.05 #parameter for the exponential smoother
MC <- 10000 #number of Monte-Carlo repetitions
X <- matrix(rnorm(n*MC), n, MC)

system.time(vol <- vol.exp.smooth.r(X = X, lambda = lambda))
```

takes

```
user  system elapsed
 34.173   0.099  34.288
```

which is not that quick even though n and MC are rather moderate in this example.

EWEA: R code using multiple cores

In the previous example, we used just one core for computations. We can easily run computations in parallel, using all available cores. First we need to start a cluster of multiple R instances.

```
\require(parallel) #R package for parallel computations
no.cores <- 8 #depending on machine (usually 4 to 8)
cl <-  makeCluster(no.cores) #this runs no.core instances of R
```

Next we slightly modify the `vol.exp.smooth.r` function.

```
vol.exp.smooth.rpcl <- function(X, lambda, cl, ...){
  clusterExport(cl, list("lambda")) #make sure that all nodes know lambda
  parApply(cl, X, 2, function(x){ #parallel version of apply
    n <- length(x)
    vol <- rep(0, n)
    vol[1] <- x[1]^2
    for(j in 2:n) vol[j] <-  lambda * x[j]^2 + (1-lambda) * vol[j-1]
    return(vol)
  })
}
```

EWEA: Execution time of the R code using multiple cores

- In theory, we should get 8-fold speed-up in our example. In reality,

  ```
  system.time(vol <- vol.exp.smooth.rpcl(X, lambda, cl))
  ```

  takes

  ```
     user  system elapsed
    2.777   0.583  11.436
  ```

  which is roughly 3 times quicker.
- This is due to the fact that the nodes need to be synchronised and as a results **some of the computations are run sequentially, not in parallel**.

EWEA: Code profiling with `Rprof`

To identify bottlenecks in our code, we can use R profiling tools.

```
Rprof() #start profiling
vol <- vol.exp.smooth.r(X = X, lambda = lambda)
Rprof(NULL) #stop profiling
summaryRprof()
```

Not surprisingly, computation of the volatility takes the majority of time.

	self.time	self.pct	total.time	total.pct
"FUN"	31.70	87.52	35.66	98.45
"^"	1.48	4.09	1.48	4.09
"-"	1.10	3.04	1.10	3.04
"*"	0.82	2.26	0.82	2.26
"+"	0.28	0.77	0.28	0.77
"("	0.26	0.72	0.26	0.72
"apply"	0.22	0.61	36.22	100.00
"array"	0.22	0.61	0.22	0.61
"aperm.default"	0.08	0.22	0.08	0.22
"unlist"	0.04	0.11	0.04	0.11
":"	0.02	0.06	0.02	0.06

EWEA: Writing C extensions for R

- R is an interpreted language with dynamic typing, therefore functions written in R are not optimal.
- We can write R extensions in low-level programming languages such as C/C++/Fortran to significantly speed-up our functions.
- In our example it is actually quite easy!

EWEA: C code

We start from writing C function calculating $\hat{\sigma}_t$ for all $t = 1, \ldots, n$.

```
#include <R.h>

void vol_exp_smooth(double *x, int *n, double *lambda, double *vol){

  unsigned int i = 0;
  vol[0] = x[0]*x[0];
  for(i=1; i<(*n); i++)
   vol[i] = (*lambda) * (x[i-1] *x[i-1]) + (1-(*lambda))*vol[i-1];

}
```

EWEA: R wrapper for the C code

Then we write an R wrapper for the C function.

```
vol.exp.smooth.rpc <- function(X, lambda, ...){
  apply(X, 2, function(x){
    res <- .C("vol_exp_smooth", as.double(x), as.integer(length(x)),
     as.double(lambda), as.double(rep(0,length(x))))
    vol <- res[[4]]
    return(vol)
  })
}
```

The following command can be used to compile the C function.

```
#this line produces vol_exp_smooth.so (Linux)
#or vol_exp_smooth.dll (Windows) file
R CMD SHLIB vol_exp_smooth.c  -fopenmp #produces
```

EWEA: Running C code in R

To use our C function in R, we need to make sure is loaded.

```
if(!is.loaded("vol_exp_smooth")) dyn.load("../lib/vol_exp_smooth.so")
system.time(vol <- vol.exp.smooth.rpc(X = X, lambda = lambda))
```

The code is 44 time quicker than the single-core R version!

```
   user  system elapsed
  0.747   0.076   0.823
```

EWEA: Performance study

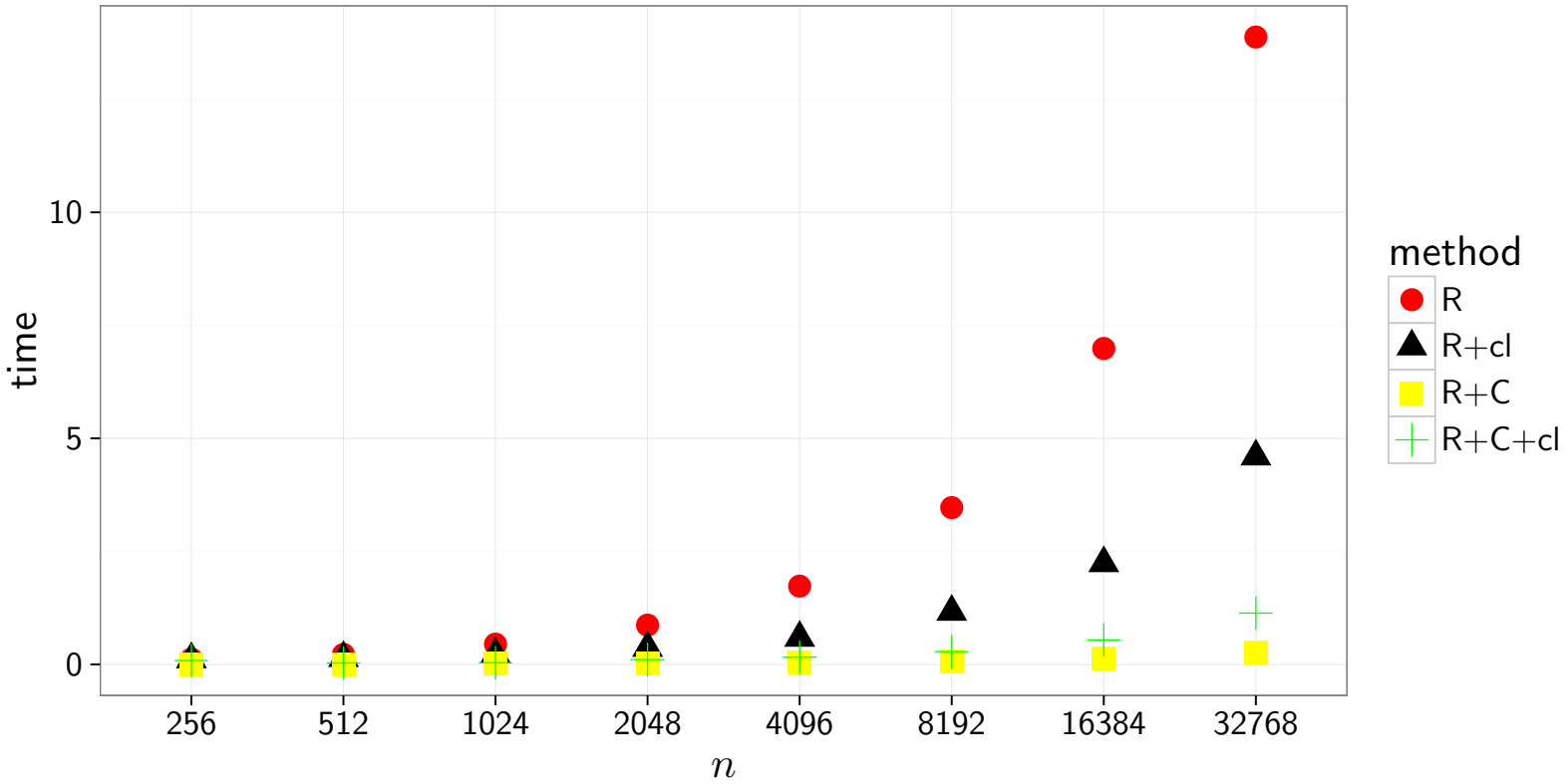

Figure : Performance of the discussed implementations of the exponential smoothing with $MC = 256$

EWEA: Performance study

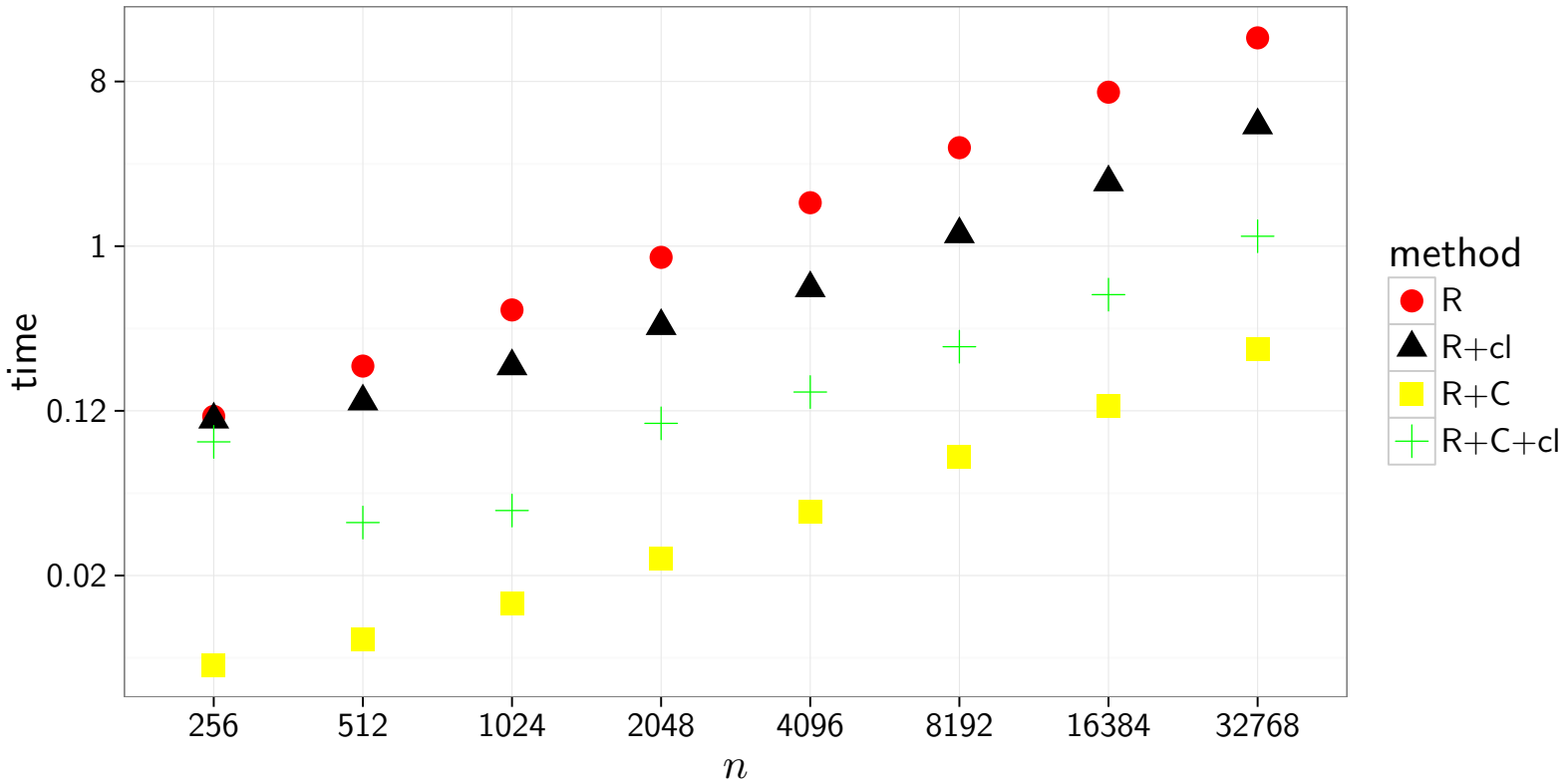

Figure : Performance of the discussed implementations of the exponential smoothing $MC = 256$ (logarithmic scale)

Speeding up Linear Algebra: Introduction

How does R perform linear algebra?

- Simple operations (vector-vector addition, matrix-vector or matrix-matrix multiplication, etc.) are implemented in a BLAS library (Basic Linear Algebra Subprograms).
- More advanced linear algebra (QR decomposition, matrix inverse, etc.) are implemented in LAPACK (LAPACK Linear Algebra PACKage).
- BLAS and LAPACK are written in Fortran. R calls their functions internally.

Speeding up Linear Algebra: Introduction

- There is a number of various implementations of BLAS and LAPACK routines more optimal than ATLAS used by R.
  - OpenBLAS (BLAS and LAPACK, even more support for parallel computations, free)
  - IntelMKL (BLAS and LAPACK, parallel computations, optimised for Intel CPU's, paid license required)
  - NVBLAS (BLAS only, performs computations on a GPU, requires CUDA capable-device, free)
- **Replacing the standard libraries with one of the above can result in a substantial speed-up of R code**.

Speeding up Linear Algebra: Matrix multiplication example

We test standard BLAS, OpenBLAS using the following example.

```
n <- 512

A <- matrix(rnorm(n^2), n, n)
B <- matrix(rnorm(n^2), n, n)

system.time(C <- A%*%B)
```

Speeding up Linear Algebra: Matrix multiplication example

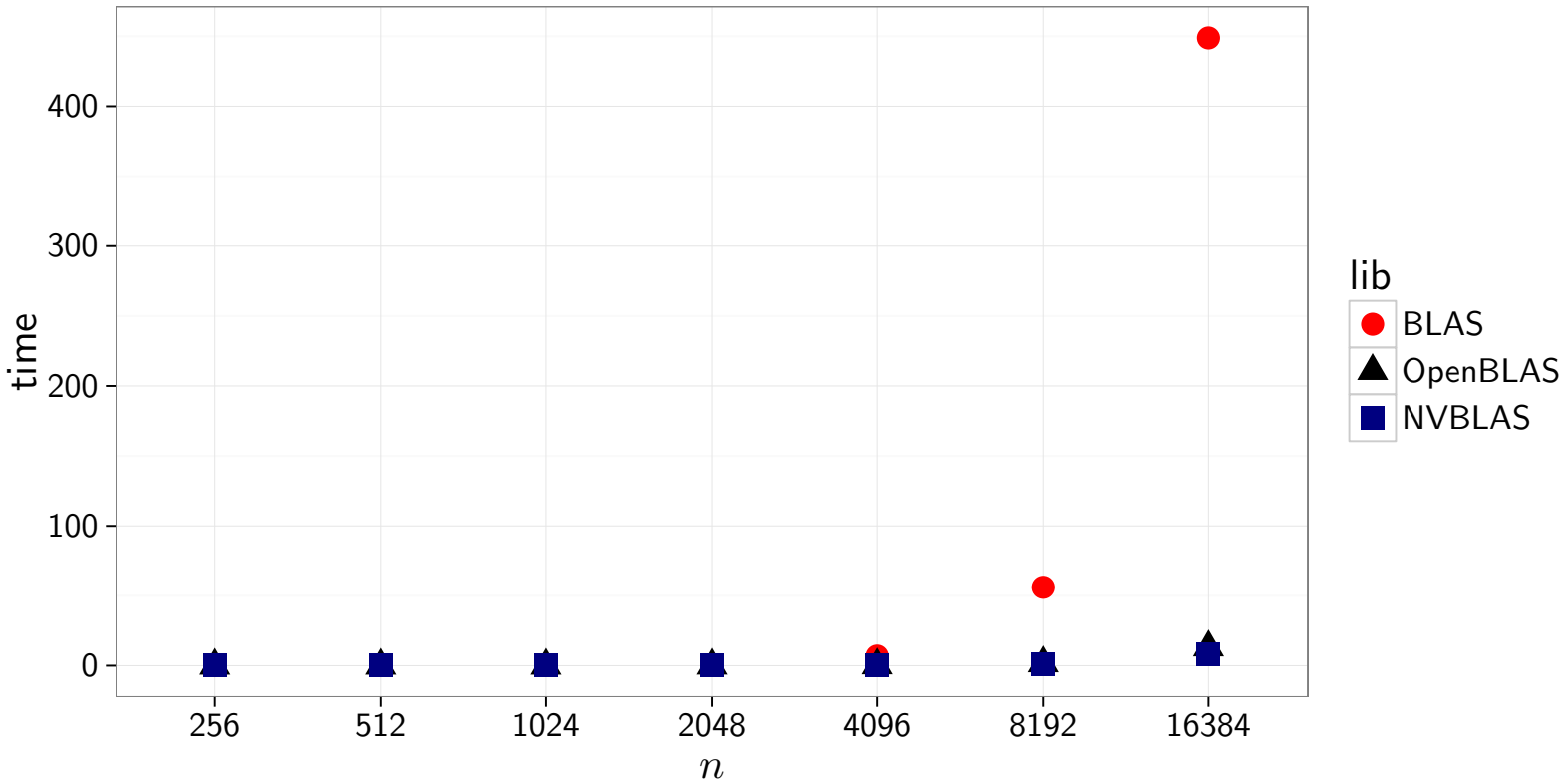

Figure : Performance of matrix multiplication for various implementations of BLAS [Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz, NVidia Quadro 4000]

Speeding up Linear Algebra: Matrix multiplication example

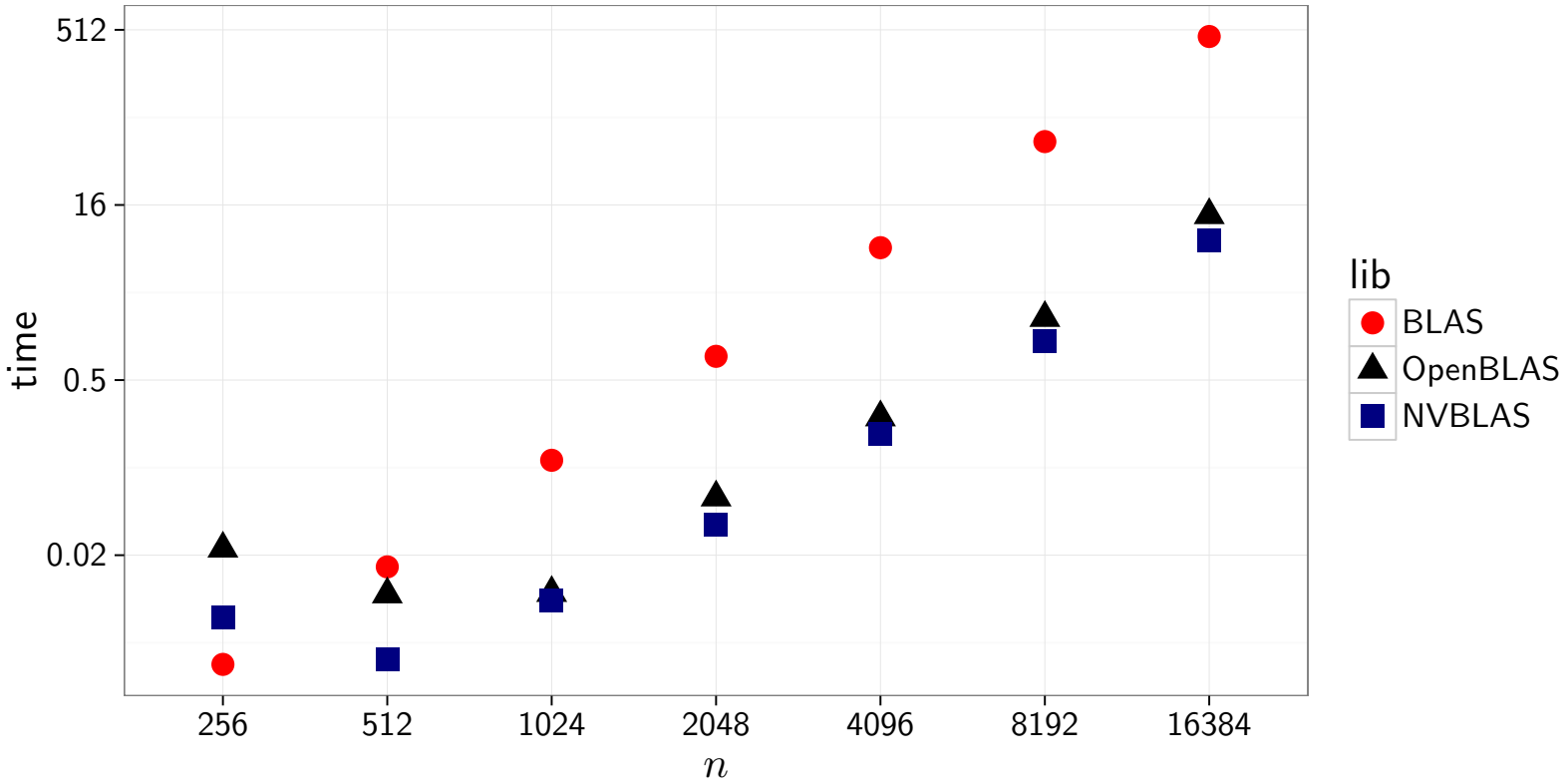

Figure : Performance of matrix multiplication for various implementations of BLAS (logarithmic scale)[Intel(R) Xeon(R) CPU E5-1620 0 @ 3.60GHz, NVidia Quadro 4000]

Resources

- R and C code and this presentation (http://personal.lse.ac.uk/baranows/).
- An OpenBLAS-based Rblas for Windows 64: Step-by-step (http://www.r-bloggers.com/an-openblas-based-rblas-for-windows-64-step-by-step/).
- For faster R use OpenBLAS instead: better than ATLAS, trivial to switch to on Ubuntu (http://www.r-bloggers.com/for-faster-r-use-openblas-instead-better-than-atlas-triv